(4)

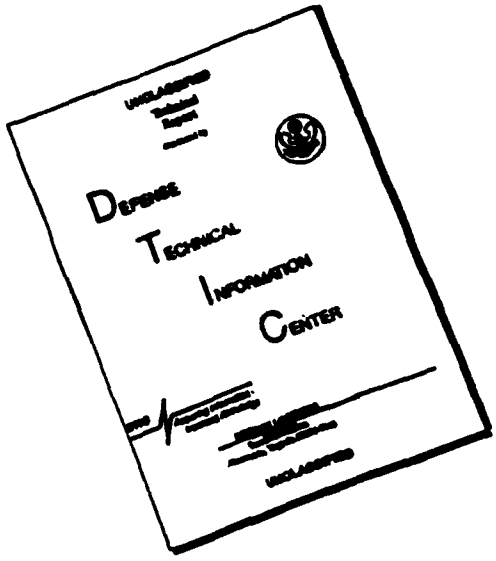
MIT/LCS/TR-437

FINDING FARTHEST NEIGHBORS IN A CONVEX POLYGON AND RELATED PROBLEMS

Dina Kravets

January 1989

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-437	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-80-C-0622
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Finding Farthest Neighbors in a Convex Polygon and Related Problems

12. PERSONAL AUTHOR(S)
Kravets, Dina — Thesis

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Master's	FROM ____ TO ____	1989 January	26

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computational Geometry, Convex Polygon, Farthest Neighbors, Monotone Matrices, Sorting

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Aggarwal et al. (A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, R. Wilber, "Geometric Applications of a Matrix-Searching Algorithm," Algorithmica, Vol. 2, 1987, pp. 195-208) showed how to compute in O(n) time one farthest vertex for every vertex of a convex n-gon. This thesis extends the results of Aggarwal et. al. by developing the following algorithms:

1. An optimal algorithm to find all farthest vertices for every vertex of a convex polygon.

2. An O(knlogk) time algorithm to find k farthest vertices for every vertex of a convex n-gon.

3. An $O(n^2)$ algorithm to sort the distances of all the vertices of a convex n-gon

(continued on back)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

GAO. Government Printing Office 1988-607-047

Unclassified

1.A

with respect to each vertex of the convex n-gon.

4. A worst-case optimal algorithm to sort a set of numbers given lower bounds on the ranks.

Finding Farthest Neighbors in a Convex Polygon and Related Problems

by

Dina Kravets

B.S.E., Electrical Engineering and Computer Science
Princeton University
(1987)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
December 1988

Signature of Author _____
Department of Electrical Engineering and Computer Science
January 24, 1989

Certified by _____
Alok Aggarwal
Visiting Professor of Mathematics
Thesis Supervisor

and by _____
Frank Thomson Leighton
Associate Professor of Mathematics
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Finding Farthest Neighbors in a Convex Polygon and Related Problems

by

Dina Kravets

Submitted to the
Department of Electrical Engineering and Computer Science
on January 24, 1989
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Aggarwal et al. [A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, R. Wilber, "Geometric Application of a Matrix-Searching Algorithm," Algorithmica, Vol. 2, 1987, pp. 195-208] showed how to compute in $O(n)$ time *one farthest vertex* for every vertex of a convex n-gon. This thesis extends the results of Aggarwal et. al. by developing the following algorithms:

1) An optimal algorithm to find *all farthest vertices* for every vertex of a convex polygon;
2) An $O(kn \log k)$ time algorithm to find k *farthest vertices* for every vertex of a convex n-gon;
3) An $O(n^2)$ algorithm to sort the distances of all the vertices of a convex n-gon with respect to each vertex of the convex n-gon; and
4) A worst-case optimal algorithm to sort a set of numbers given lower bounds on the ranks.

Thesis Supervisor: Alok Aggarwal

Title: Visiting Professor of Mathematics

Thesis Supervisor: Frank Thomson Leighton

Title: Associate Professor of Mathematics

Keywords: Computational Geometry, Convex Polygon, Farthest Neighbors, Monotone Matrices, Sorting.

Portions of this thesis are joint work with Alok Aggarwal and James Park. The author was supported in part by Air Force Contract OSR-86-0076, DARPA Contract N00014-80-C-0622, and Army Contract DAAL-03-86-K-0171.

1 Introduction

Given a set S of n points in the plane, for each $p_i \in S$ we want to find a point $p_j \in S$, $j \neq i$, such that

$$d(p_i, p_j) = \max_{0 \leq l \leq n-1} d(p_i, p_l),$$

where $d(p_i, p_j)$ is the Euclidean distance between p_i and p_j. We will call this the *farthest neighbor* problem. Toussaint and Bhattacharya [TB81], and Preparata [P77] have shown a $\Theta(n \log n)$ time bound for this problem. However, an $\Omega(n \log n)$ time bound is obtained under the assumption that the input to the algorithm is an arbitrary set of points in the plane. If we restrict the input to be the vertices of a convex polygon, then Aggarwal et. al. [AKMSW87] have shown a $\Theta(n)$ time bound. In section 2 we define monotone matrices that are used in [AKMSW87] to solve the farthest neighbor problem. We use these matrices to obtain our results as well.

We extend their results in the following ways:

1. The *all farthest neighbors* problem for a set S of n points is to find for every point $p_i \in S$, *all* the points p_j, $j \neq i$, such that

   $$d(p_i, p_j) = \max_{0 \leq l \leq n-1} (p_i, p_l).$$

   In section 3.1 we show how to find all farthest neighbors for every vertex of a convex polygon and in section 3.2 we prove $\Theta(n)$ time complexity for our algorithm. In section 3.3 we give an application of our algorithm.

2. The *k farthest neighbors* problem for a set S of n points is to find, for every point $p_i \in S$, the set of points P_i, $|P_i| = k$, such that every $p_j \in P_i$ is farther from p_i than any $p_l \notin P_i$. In section 4.1 we show how to find k farthest neighbors for every vertex of a convex polygon and in section 4.2 we prove $O(kn \log k)$ time complexity for our algorithm. Section 4.3 contains some comments on the lower bound for this problem.

3. The *sorting problem for a planar point set* is as follows: given two sets S and T containing n and m planar points, respectively, sort the points of T with respect to their distance from p, for each $p \in S$. If $T = S$, then the problem reduces to sorting S with respect to itself.

   Clearly, we can sort T with respect to S in $O(nm \lg m)$ time, using standard sorting techniques[1]. This naive approach, however, is not optimal when S is a convex set, i.e. it contains the vertices of a convex polygon. In Section 5.1, we show how to sort T with respect to S if S is a convex set. In section 5.2 we prove $O(nm + m^2)$ time complexity for our algorithm, which implies that

[1] [CLR89], part II.

we can sort S with respect to itself in $O(n^2)$ time. Sorting a convex set S with respect to itself corresponds to the case $k = n$ in the k farthest neighbors problem if we require that the list of farthest neighbors be produced in sorted order.

Section 5.3 contains some comments on the lower bound for this problem.

We extend this result in Section 5.4, obtaining an $O(n^2 \log \ell)$ time algorithm for sorting an arbitrary point set S (with ℓ convex layers) with respect to itself.

In addition to the problems concerning finding farthest neighbors, in section 6 we show how to sort a set of numbers given a lower bound on the rank of each number in the sorted order. We give an algorithm that solves this problem in section 6.2, find the time complexity of the algorithm in section 6.3, and prove that it is optimal in the worst case in section 6.4. This problem is related to other questions of sorting using partial information which are investigated in, for example, [BT80], [F76], [GMPR77], [KS84], [LS83].

All the lower bounds in this paper are given in the *algebraic decision tree* model of computation[2]. Briefly, in this model the cost of running the algorithm is considered to be proportional to the number of comparisons along the longest path from the root to a leaf in the decision tree that corresponds to the execution of the algorithm.

[2] For detailed discussion, see page 30 of [PS85].

2 Monotone Matrices

In this section we explain the construction used in [AKMSW87] and by us to solve the farthest neighbor problems. A two-dimensional matrix $M = \{m_{i,j}\}$ is called *monotone* if the maximum value in the i-th row lies below or to the right of the maximum value in the $(i-1)$-st row. If a row has several maxima then we will take the leftmost one. A matrix M is called *totally monotone* if every 2×2 submatrix (i.e., every 2×2 minor) is monotone (See Figure 1).

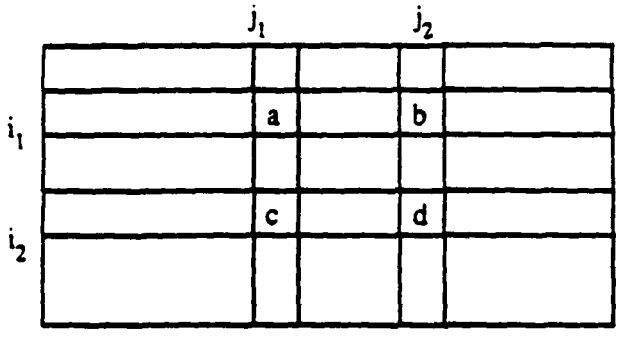


Figure 1: Every 2x2 minor , given by a,b,c, and d, of a totally monotone matrix is monotone, i.e. it is not possible that a<b and c>d.

Although the question of finding the row maxima in a two-dimensional totally monotone matrix may seem rather odd at first glance, [AKMSW87] and [AP88] show that a wide variety of problems can be reduced to one or more instances of this problem. For example, the totally monotone property arises in problems that deal with polygons whose vertices obey the quadrangle inequality. Suppose a convex polygon has n vertices numbered $0, 1, \ldots, n-1$. If we take any 4 distinct vertices, i_1, i_2, j_1, and j_2, such that $0 \le i_1 \le i_2 \le j_1 \le j_2 \le n-1$, and form a quadrangle $i_1 i_2 j_1 j_2$, then by the quadrangle inequality, the sum of lengths of the diagonals $d(i_1, j_1) + d(i_2, j_2)$ is *strictly greater* than the sum of the lengths of the sides $d(i_1, j_2) + d(i_2, j_1)$ (See Figure 2).

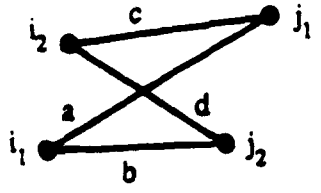


Figure 2

We define a matrix $M = \{m_{i,j}\}$ corresponding to a convex polygon as follows

(See Figure 3): $\forall i,\ 0 \le i \le n-1,\ m_{i,j} = d(i, j \bmod n))$ for $i < j < i + n$, and $m_{i,j} = -\infty$ for all other j.

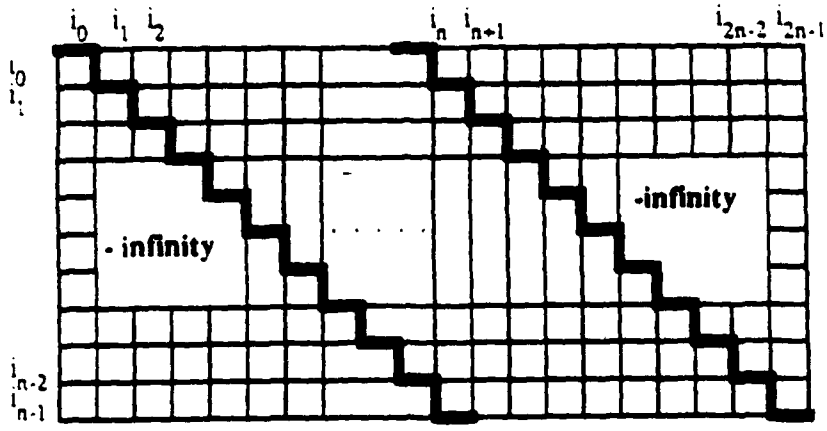


Figure 3

Claim 1 *Matrix M is a totally monotone matrix.*

Proof Take any 2×2 minor (See Figure 1). If none of the entries in the minor are $-\infty$ and $i_1 < i_2 < j_1 < j_2$, then it is not possible that $a < b$ and $c > d$ because of the quadrangle inequality. $i_1 < i_2$ and $j_1 < j_2$ have to hold simply from the way we have drawn the minor. If $i_2 \ge j_1$, then $c = -\infty$. Similarly, if any other inequality among $i_1 < i_2 < j_1 < j_2$ does not hold, then at least one of the entries in the minor is $-\infty$, which is the case we consider next.

If $b = -\infty$ or $c = -\infty$, then either $a \not< b$ or $c \not> d$. If $a = -\infty$, then by the way we have defined M, either $b = -\infty$ or $c = -\infty$ and the previous case applies. Similarly, if $d = -\infty$, then either $c = -\infty$ or $b = -\infty$. Thus, M is monotone. ■

If each entry of a totally monotone $n \times m$ matrix M can be computed in constant time, then Aggarwal et al. [AKMSW87] showed how to find the leftmost maximum in each row of an $n \times m$ matrix M in $O(n + m)$ time on a sequential RAM. In the following discussion we will refer to the [AKMSW87] algorithm as the SMAWK algorithm, following the convention used by the authors. Note that the SMAWK algorithm does not explicitly create the entire matrix M (that would take $O(nm)$ time): rather, it only computes $O(n + m)$ entries of M. Thus, the [AKMSW87] algorithm can find the *leftmost farthest neighbor* of every vertex on a convex n-gon in $O(n)$ time.

Before concluding this section, we note that if f_i denotes the number of farthest neighbors of the i-th vertex, then it can be shown (see appendix) that $\sum_{i=0}^{n-1} f_i \le 2n$.

Observe that there are convex polygons for which $\sum_{i=0}^{n-1} f_i = 2n$. For example, put $n-1$ vertices on an arc such that there are vertices at both endpoints of the arc and the size of the arc is $\frac{1}{6}$ of a circle. Let the center of the circle be the n-th vertex. The center has $n-1$ farthest neighbors, each endpoint of the arc has 2 farthest neighbors, and each of the other $n-3$ points on the arc has 1 farthest neighbor. The sum is $(n-1)+2\times 2+(n-3)\times 1 = 2n$.

Two vertices i and j are called *symmetric farthest neighbors* if $f(i) = j$ and $f(j) = i$. Since $\sum_{i=0}^{n-1} f_i \leq 2n$, there are at most n symmetric farthest neighbor pairs. The polygon described above again achieves the upper bound of n.

3 The All Farthest Neighbors Problem for a Convex Polygon

3.1 The Algorithm for Finding All Farthest Neighbors of a Convex Polygon

Let $M = \{m_{i,j}\}$ be a totally monotone $n \times 2n$ matrix corresponding to a convex polygon that was defined in section 2. We use the SMAWK algorithm to find the leftmost maximum in each row of M. Now, consider a different totally monotone matrix M' such that $\forall i,\ 0 \leq i \leq n-1,\ m'_{i,j} = d((n-1-i) \bmod n, (2n-1-j) \bmod n)$ for $i < j < i+n$ and $m'_{i,j} = -\infty$ for all other j.

Note that the matrix M' is simply M rotated upside down, or equivalently, M flipped horizontally and vertically (See Figure 4).

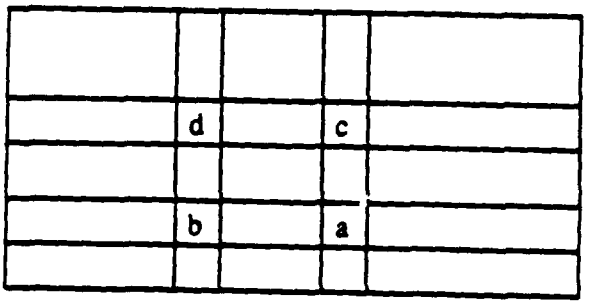


Figure 4: The matrix from Figure 1 flipped horizontally and vertically.

Such a rotation preserves the totally monotone property: it is not possible to have $d < c$ and $b > a$ (which is the same as saying that it is not possible to have $a < b$ and $c > d$). Again, we use the SMAWK algorithm to find the leftmost maximum in each row of M'. But, by the way we have defined M', the leftmost maximum in row i of M' is the rightmost maximum in row $(n-1-i) \bmod n$ of M. Therefore, we now have the rightmost maximum in each row of M. Last, for each row i, check all the entries between the leftmost and the rightmost maxima in i for other maxima in that row. The summary of the algorithm follows.

This section is joint work with Alok Aggarwal.

```
run SMAWK on M
for i — 1 to n do
    l_i — column of the leftmost maximum ... row i of M
    output (i, l_i)
run SMAWK on M'
for i — 1 to n do
    r_i — column of the leftmost maximum in row (n − 1 − i) mod n of M'
    if l_i ≠ r_i then output (i, r_i)
for i — 1 to n do
    for j — l_i + 1 to r_i − 1 do
        if m_{i,j} = m_{i,l_i} then output (i, j)
```

3.2 Time Complexity of the Algorithm

The first two loops of the algorithm and the two executions of the SMAWK algorithm take $O(n)$ time. Clearly, all the row maxima have to be between the leftmost and the rightmost maxima in that row. Note that $m_{i,l_i} = m_{i,r_i}$. All we need to prove now is that the third loop of the algorithm takes $O(n)$ time.

From the quadrangle inequality we know that $a+d > b+c$ (See Figure 2). From this we derive the following observations:

$$a \leq b \text{ implies } c < d$$
$$c \geq d \text{ implies } a > b.$$

Lemma 1 $r_i \leq min\{2n-1, l_{i+1}\}$.

Proof Suppose $r_i > min\{2n-1, l_{i+1}\}$. Let $x = m_{i,r_i} = m_{i,l_i}$, $y = m_{i,l_{i+1}}$, $z = m_{i+1,l_{i+1}}$, and $w = m_{i+1,r_i}$ (See Figure 5).

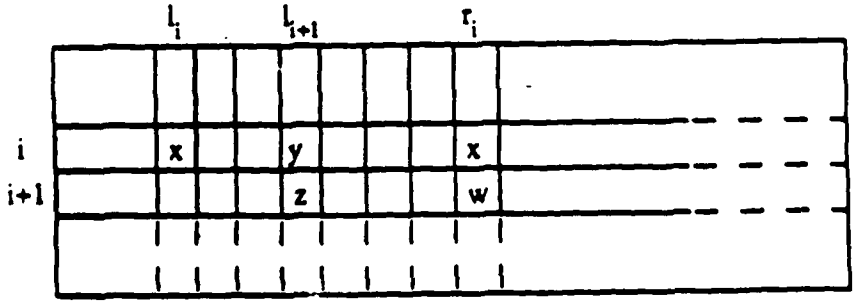


Figure 5

Since x is the maximum value for row i, it must be true that $y \leq x$. Then, by our first observation, $z < w$, which contradicts the fact that z is the maximum for row $i+1$. Thus, $r_i \leq min\{2n-1, l_{i+1}\}$. ■

Lemma 2 *The third loop of the algorithm takes $O(n)$ time.*

Proof Let $\mathcal{R}_i$ be the region between m_{i,l_i} and m_{i,r_i}. By Lemma 1, $\forall i, j,\ i \neq j$, $\mathcal{R}_i$ and $\mathcal{R}_j$ cannot overlap horizontally. Thus, each x-coordinate of matrix M will be checked in at most one such region in the third loop of the algorithm. Since the horizontal dimension of M is $2n$, we need to check at most $O(n)$ entries of M. ■

Since we output at least one farthest neighbor for each of the n vertices, the $O(n)$ running time of our algorithm is optimal.

3.3 An Application: All Symmetric Farthest Neighbors of a Simple Polygon

It is well known that the farthest neighbor of any point inside a convex region is one of the vertices on the convex hull. Also, the convex hull of a simple polygon can be found in $O(n)$ time ([GY83]). So, to find all symmetric farthest neighbors of a simple polygon we first find the convex hull of the simple polygon. Then, using the algorithm in section 3.1, we find all farthest neighbors of the convex hull, and finally scan the list of vertices of the convex hull to find which ones are symmetric farthest neighbors. Thus, all symmetric farthest neighbors of a simple polygon can be found in $\Theta(n)$ time. This settles a problem raised in [T83].

4 The k Farthest Neighbors Problem for a Convex Polygon

4.1 The Algorithm for Finding k Farthest Neighbors of a Convex Polygon

Let $M = \{m_{i,j}\}$ be a totally monotone $n \times m$ matrix. Our algorithm consists, roughly speaking, of three parts. In the first and second parts, we decompose each row of M as follows: for each row j, we partially order elements of row j into $2k$ sorted lists each with upto k elements such that the k largest elements of row j are guaranteed to be among the lists for row j[1]. In the third part, for each row j we find the k largest elements in j. We achieve this by partially merging the $2k$ lists for j that we found in the first two parts.

Let $c[i,j]$ be the column of the largest element in the i-th sorted list for row j. The first part has k phases. In the i-th phase we find $(2i-1)$-st and $2i$-th list of upto k elements for each row j using the $2(i-1)$ lists found in the previous phases. We achieve this by running the SMAWK algorithm on the matrix M', where M' is M with columns $c[i',j']$ for $1 \leq i' \leq i-1$ and $1 \leq j' \leq n$ removed. Running the SMAWK algorithm gives us $c[i,j]$ for each row j. Moreover, because M and M' are totally monotone, $c[i,j]$'s actually give us more information. If we look at the matrix $\hat{M}$ given by the columns $c[i,1]$ through $c[i,n]$,[2] we observe that the largest element in row j of $\hat{M}$ is $m_{j,c[i,j]}$. Furthermore, each row of $\hat{M}$ is bitonic.

Lemma 3 *$\forall j$ and $\forall i$,*

$$m_{j,c[i,1]} \leq \cdots \leq m_{j,c[i,j-1]} \leq m_{j,c[i,j]} \text{ and}$$

$$m_{j,c[i,j]} \geq m_{j,c[i,j+1]} \geq \cdots \geq m_{j,c[i,m]}.$$

Proof Suppose not, i.e., there is some i and some row j of $\hat{M}$ such that for some s and t, $t < s < j$, $m_{j,c[i,s]} > m_{j,c[i,s]}$ (the case of $j < s < t$ follows analogously). If $c[i,s] = c[i,t]$, then certainly $m_{j,c[i,t]} = m_{j,c[i,s]}$. If $c[i,s] \neq c[i,t]$, then since $m_{s,c[i,s]}$ is the largest element in row s, $m_{s,c[i,t]} \leq m_{s,c[i,s]}$. But if M is totally monotone, then it is impossible that $m_{s,c[i,t]} < m_{s,c[i,s]}$ and $m_{j,c[i,t]} > m_{j,c[i,s]}$. Thus, our assumption must have been wrong. ■

Thus, for each row j, the $c[i,j]$'s define $2k$ sorted lists:

This section is joint work with James Park.

[1] Note that we seem to be dealing with $O(k^2 n)$ elements in these lists. We only use these lists for the explanation; as we shall see, the algorithm actually uses only $O(kn)$ elements of M.

[2] Note that $c[i,j]$ may equal $c[i,j+1]$ for some j's, but each column of M appears only once in $\hat{M}$.

$$
\begin{aligned}
&m_{j,c[1,j]} \geq m_{j,c[1,j-1]} \geq \cdots \geq m_{j,c[1,1]} \\
&m_{j,c[1,j]} \geq m_{j,c[1,j+1]} \geq \cdots \geq m_{j,c[1,m]} \\
&m_{j,c[2,j]} \geq m_{j,c[2,j-1]} \geq \cdots \geq m_{j,c[2,1]} \\
&m_{j,c[2,j]} \geq m_{j,c[2,j+1]} \geq \cdots \geq m_{j,c[2,m]} \\
&\vdots \\
&m_{j,c[k,j]} \geq m_{j,c[k,j-1]} \geq \cdots \geq m_{j,c[k,1]} \\
&m_{j,c[k,j]} \geq m_{j,c[k,j+1]} \geq \cdots \geq m_{j,c[k,m]}.
\end{aligned}
$$

Unfortunately, we run into a problem here since $c[i,j']$ may be equal to $c[i,j'+1]$ for some number of (j')s. This is a problem because in order to achieve the desired time bounds we want each member of each list to be a different entry of M. To solve the difficulty we need to "skip" the repeating entries. This is accomplished in the second part by defining the skip variables

$$up[i,j] = j - |\{j' : 1 < j' < j \text{ and } c[i,j'] = c[i,j]\}| - 1$$

$$down[i,j] = j + |\{j' : n > j' > j \text{ and } c[i,j'] = c[i,j]\}| + 1$$

Now, we can define $2k$ sorted lists for each row j in which no entry of M is repeated:

$$
\begin{aligned}
&m_{j,c[1,j]} \geq m_{j,c[1,up[1,j]]} \geq m_{j,c[1,up[1,up[1,j]]]} \geq \cdots \\
&m_{j,c[1,j]} \geq m_{j,c[1,down[1,j]]} \geq m_{j,c[1,down[1,down[1,j]]]} \geq \cdots \\
&m_{j,c[2,j]} \geq m_{j,c[2,up[2,j]]} \geq m_{j,c[2,up[2,up[2,j]]]} \geq \cdots \\
&m_{j,c[2,j]} \geq m_{j,c[2,down[2,j]]} \geq m_{j,c[2,down[2,down[2,j]]]} \geq \cdots \\
&\vdots \\
&m_{j,c[k,j]} \geq m_{j,c[k,up[k,j]]} \geq m_{j,c[k,up[k,up[k,j]]]} \geq \cdots \\
&m_{j,c[k,j]} \geq m_{j,c[k,down[k,j]]} \geq m_{j,c[k,down[k,down[k,j]]]} \geq \cdots
\end{aligned}
$$

Also, observe that $\forall j$, $m_{j,c[1,j]} \geq m_{j,c[2,j]} \geq \cdots \geq m_{j,c[k,j]}$. In other words, $m_{j,c[i',j]}$ could be the i-th largest in row j only if $m_{j,c[i'-1,j]}$ is one of the $i-1$ largest.

In the third part of our algorithm, for each row j of M we merge parts of these lists to get the k largest elements. This part of the algorithm consists of k phases (for each row j). In phase i we find i-th largest value in row j using $i-1$ largest values in that row. We partially merge these lists using a data structure H. H must support the following operations: insert, find the maximum element and delete the maximum element. One suitable implementation for H is a heap[3]. The algorithm is summarized below.

[3]For detailed discussion, see chapter 7 of [CLR89].

```
M' ← M
for i ← 1 to k do
    run SMAWK on M'
    for j ← 1 to n do
        c[i, j] ← column of the maximum in row j of M'
    for j ← 1 to n do
        M' ← M' - column c[i, j]
for i ← 1 to k do
    down ← 1; up ← 1; high ← 1
    while high < n do
        low ← high
        while c[i, high] = c[i, high + 1] and high < n do
            high ← high + 1
        for j ← low to high do
            up[i, j] ← low − 1
            down[i, j] ← high + 1
        high ← high + 1
for j ← 1 to n do
    L[1, j] ← m_{j,c[1,j]}
    initialize H
    insert(H, m_{j,c[1,down[1,j]]}, m_{j,c[1,up[1,j]]}, m_{j,c[2,j]})
    for i ← 2 to k do
        m_{j,c[i',j']} ← find-max(H)
        L[i, j] ← m_{j,c[i',j']}
        delete-max(H)
        if j' ≤ j then insert(H, m_{j,c[i',up[i',j']]})
        if j' ≥ j then insert(H, m_{j,c[i',down[i',j']]})
        if j' = j then insert(H, m_{j,c[i'+1,j]})
```

4.2 Time Complexity of the Algorithm

The first part of the algorithm is running the SMAWK algorithm k times and thus takes $O(k(n+m))$ time. The second part does a constant number of operations for each $c[i,j]$, thus taking $O(k(n+m))$ time. In the third loop, for each i, we are running k insert, find-max and delete-max operations on a heap H. Each of these operations takes no more than $O(\log|H|)$ time, where $|H|$ is the number of elements in H at the time the operation is performed. Luckily, H never has too many elements. In fact, $|H| \leq 2k$ since at any point H contains at most one element from each of the $2k$ lists. Thus, the second part of the algorithm takes $O(kn\log k)$ time and the total time for the algorithm is $O(kn\log k)$.

Note that this algorithm could be used to find k farthest neighbors on a convex n-gon in $O(kn\log k)$ time.

4.3 Some Comments on the Lower Bounds

To output n lists of k sorted elements each would seem to require $\Omega(kn \log k)$ time. However, these lists are not necessarily independent. In terms of our monotone matrix M, these lists are not necessarily column independent, i.e., one of the k largest elements in row i may be in the same column as one of the k largest elements in row i'. In terms of a polygon, the same vertex could be included as one of the k farthest vertices for more than one vertex. Yet, no $o(kn \log k)$ time algorithm is known that gives the k largest elements even when the elements are not required in sorted order.

5 The Sorting Problem for a Planar Point Set

5.1 The Algorithm for Sorting the Points of an Arbitrary Set T with respect to a Convex Set S

Let a convex set S be given by the vertices of a convex polygon P_S. Let $\{s_0, \ldots, s_{n-1}\}$ be the points of S in clockwise order around the perimeter of P_S and let $\{t_0, \ldots, t_{m-1}\}$ be the points of T in any order. Denote by $m_{i,j}$ the distance from s_i to t_j and by $j[i,r]$ the index of the point in T whose distance from s_i is r-th smallest among points in T. (For convenience, we define $j[0,r] = r$ for $1 \leq r \leq m$.)

Our algorithm consists of n phases, where in the i-th phase, we compute $j[i,r]$ for $1 \leq r \leq m$, using the values $j[i-1,r]$ we computed in the previous phase. We do this using an insertion sort, inserting $t_{j[i-1,1]}$, then $t_{j[i-1,2]}$, then $t_{j[i-1,3]}$, and so on through $t_{j[i-1,m]}$. This is summarized below.

```
for r ← 1 to m do
    j[0,r] ← r
for i ← 0 to n − 1 do
    for r ← 1 to m do
        j[i,r] ← j[i − 1,r]
        k ← r
        while k > 1 and m_{i,j[i,k]} > m_{i,j[i,k−1]} do
            t ← j[i,k]
            j[i,k] ← j[i,k − 1]
            j[i,k − 1] ← t
            k ← k − 1
```

5.2 Time Complexity of the Algorithm

We claim that the i-th phase of our algorithm requires $O(m+I_i)$ time, where I_i is the number of inversions that occur in going from the ordering of T's points by distance from s_{i-1} to the ordering of T's points by distance from s_i. An inversion corresponds to a pair of points $(t_j, t_{j'})$ from T such that $m_{i-1,j} \leq m_{i-1,j'}$ but $m_{i,j} \geq m_{i,j'}$. Since the perpendicular bisector of any pair of points $(t_j, t_{j'})$ from T intersects P_S at most twice,

$$\sum_{i=0}^{n-1} I_i \leq 2\binom{m}{2}.$$

Thus, our algorithm runs in $O(nm + m^2)$ total time.

This section is joint work with James Park.

Note that this algorithm can be used to sort the rows of an $n \times m$ totally monotone matrix $M = \{m_{i,j}\}$ in $\mathcal{O}(nm + m^2)$ time, as total monotonicity implies that for each pair of columns (j, j'), $j < j'$, either

(1) $\forall i\ m_{i,j} \leq m_{i,j'}$ or

(2) $\exists i,\ 0 < i < n-1$ such that $m_{i',j} \geq m_{i',j'}$ for $i' \leq i$ and $m_{i',j} \leq m_{i',j'}$ for $i' > i$.

Thus, we can sort the set S of vertices of a convex polygon with respect to itself in $O(n^2)$ time.

5.3 Some Comments on Lower Bounds

Sorting T with respect to S would seem to require $\Omega(nm)$ time. This is because specifying n orderings of m points in the obvious way, namely, by outputting n ordered lists of length m, itself requires $\Omega(nm)$ time. However, this is not necessarily the case. Certainly, we can reduce the problem of sorting m numbers to the problem of sorting m points of T with respect to one point of S. To do this reduction, we take the set of m numbers $\{x_0, \cdots, x_{m-1}\}$ and we create points $\{t_0, \cdots, t_{m-1}\}$ of T as follows: $t_i = (x_i, 0)$ for $0 \leq i \leq m-1$. Furthermore, we let $s_0 = (0,0)$. Then, sorting T with respect to s_0 is equivalent to sorting $\{x_0, \cdots, x_{m-1}\}$. Thus, we know that sorting T with respect to S requires $\Omega(m \log m)$ time. However, once we have sorted T with respect to s_0, it is not clear if sorting T with respect to s_1 requires as much time. The difficulty with showing the $\Omega(nm + m^2)$ lower bound for sorting T with respect to S is that only $O(n^4)$ differing orderings of T's points are possible — the $\binom{n}{2}$ perpendicular bisectors of pairs of points from S divide the plane into $O(n^4)$ regions, and there is a one-to-one correspondence between regions and orderings. Thus, it may be possible to specify an ordering of T's points with respect to some s_j's in $o(n)$ time, which suggests that a $o(nm)$ time solution to the problem of sorting T with respect to S is still attainable.

5.4 An $O(n^2 \lg \ell)$ Time Algorithm for Arbitrary S

We can solve the sorting problem for an arbitrary set S of n planar points using our $O(nm + m^2)$ time algorithm for sorting an arbitrary set with respect to a convex set. We use two partitions of S: we partition S into ℓ subsets $S_1, \ldots, S_\ell$, each corresponding to a convex layer of S,[1] and ℓ subsets $S'_1, \ldots, S'_\ell$, each of size n/ℓ. For $1 \leq i \leq \ell$ and $1 \leq j \leq \ell$, we sort S'_j with respect to S_i in $O(n_i n/\ell + n^2/\ell^2)$ time, where n_i is the size of S_i. The total time required is

$$\sum_{i=1}^{\ell} \sum_{j=1}^{\ell} O\left(n_i \frac{n}{\ell} + \frac{n^2}{\ell^2}\right) = \sum_{i=1}^{\ell} O\left(n_i n + \frac{n^2}{\ell}\right) = O\left(n^2\right).$$

[1] We can find convex layers of n points in $\Theta(n \log n)$ time ([PS85], p. 168).

For each point p of S, we now have ℓ sorted lists, corresponding to the points of $S'_1, \ldots, S'_\ell$, respectively. As we can merge these lists in $O(n^2 \lg \ell)$ total time, we obtain an $O(n^2 \lg \ell)$ time algorithm for sorting the points of S.

6 Sorting Using Lower Bounds on the Ranks

6.1 Background

Let $S = \{x_0, ..., x_{n-1}\}$ be a set of n numbers. To sort S requires $\Omega(n \log n)$ time and there are algorithms that achieve this lower bound ([CLR89], part II). What if we are given some partial information about S? The lower bound may not necessarily hold. Various kinds of partial information about S have been researched in [BT80], [F76], [GMPR77], [KS84] and [LS83].

We consider the following problem. Given a set S of n numbers, we have the following partial information about the final sorted order: for each $x_i \in S$, we have a lower bound l_i on x_i's rank, i.e., a lower bound on the number of $x_j \in S$ such that $x_j \leq x_i$. How much time does it take to sort S?

6.2 The Algorithm for Sorting S Using Lower Bounds on the Ranks

We create n buckets $\langle b_1, ..., b_n \rangle$. Bucket b_j will represent rank j in the sorted order of S and by the end of the algorithm, $b_j[0]$ will contain the j-th largest element of S. We put into b_j all those x_i's whose lower bound on the rank is j. Let c_j be the number of x_i's placed into b_j. We then go through the buckets one at a time, from b_1 to b_n. We use a data structure H to process the buckets. To process bucket b_j, we take the c_j elements in b_j and insert them into H. We then find the maximum element in H, put that element into $b_j[0]$ and delete it from H. H has to support the following operations: insert, find maximum element, and delete maximum element. As in section 4.1, one suitable data structure for H is a heap. The algorithm is summarized below.

```
for j ← 1 to n do
    c_j ← 0
for i ← 0 to n − 1 do
    c_{l_i} ← c_{l_i} + 1
    b_{l_i}[c_{l_i}] ← x_i
initialize H
for j ← 1 to n do
    for k ← 1 to c_j do
        insert(H, b_j[k])
    x_max ← find-max(H)
    b_j[0] ← x_max
    delete-max(H)
```

6.3 Time Complexity of the Algorithm

Let $Q = \sum_{i=1}^{n}(r_i - l_i + 1)$, where r_i is the exact rank of x_i in the sorted order[2]. Since $\langle r_1, ..., r_n \rangle$ is just a permutation of the set $\langle 1, ..., n \rangle$, $\sum_{i=1}^{n} r_i = \frac{n(n+1)}{2}$. Thus, $Q = \frac{n(n+3)}{2} - \sum_{i=1}^{n} l_i$.

Lemma 4 *The algorithm takes $O(n \log \frac{Q}{n})$ time.*

Proof The first two loops take $O(n)$ time. Note that $\sum_{j=1}^{n} c_j = n$. In step j of the third loop, we insert c_j elements into H, then find and delete the maximum in H. Each of these operations takes no more than $O(\log |H|)$ time, where $|H|$ is the number of elements in H at the time the operation is performed. So, if we let w_j be the number of elements in H at the end of the j-th iteration, then during the $(j+1)$-st iteration, we pay $O(\log(w_j) + \log(w_j + 1) + ... + \log(w_j + c_{j+1} - 1))$ to insert c_{j+1} elements into H and $O(\log(w_j + c_{j+1}))$ to find and delete the maximum. Note that our definition of the w_j's implies that $w_0 = w_n = 0$ and for $0 < j < n$, $0 \leq w_j \leq n$. We can upper-bound the time spent on the $(j+1)$-st iteration by $(c_{j+1} + 1)\log(w_j + c_{j+1})$.

Now we express the w_j's in terms of the c_j's. At the end j-th iteration of the third loop, H contains all the elements we have inserted during the first j iterations, minus the number of maxima we have deleted. Thus,

$$w_j = \sum_{i=1}^{j} c_i - j.$$

Alternatively,

$$\begin{aligned} w_j &= [\sum_{i=1}^{j-1} c_i - (j-1)] + c_j - 1 \\ &= w_{j-1} + c_j - 1. \end{aligned}$$

Then $c_j = w_j - w_{j-1} + 1$. So, the time T for the third step can be bounded as follows:

$$\begin{aligned} T &\leq O\left(\sum_{j=1}^{n}(c_j + 1)\log(w_{j-1} + c_j)\right) \\ &\leq O\left(\sum_{j=1}^{n}((w_j - w_{j-1} + 1) + 1)\log(w_{j-1} + (w_j - w_{j-1} + 1))\right) \\ &\leq O\left(\sum_{j=1}^{n}(w_j - w_{j-1} + 2)\log(w_j + 1)\right) \end{aligned}$$

[2]Note that we are only using r_i's to simplify our explanation, we do not need them at all since Q is simply a function of l_i's and n.

$$\leq O\left(w_1 \log\left(\frac{w_1+1}{w_2+1}\right)\right) + O\left(w_2 \log\left(\frac{w_2+1}{w_3+1}\right)\right) + \ldots + O\left(w_{n-1} \log\left(\frac{w_{n-1}+1}{w_n+1}\right)\right)$$
$$+O\left(w_n \log(w_n+1)\right) + O\left(\sum_{j=1}^{n} \log(w_j+1)\right).$$

The terms corresponding to any $w_j = 0$ dissappear. So, henceforth, we assume that $\forall j,\ w_j \neq 0$. Since $\forall j\ c_j \geq 0$, $w_{j-1} - 1 \leq w_j$, and thus, $\frac{1}{w_{j-1}} \geq \frac{1}{w_j+1}$. Then,

$$T \leq O\left(w_1 \log\left(\frac{w_1+1}{w_1}\right)\right) + O\left(w_2 \log\left(\frac{w_2+1}{w_2}\right)\right) + \ldots + O\left(w_{n-1} \log\left(\frac{w_{n-1}+1}{w_{n-1}}\right)\right)$$
$$+O\left(w_n \log(w_n+1)\right) + O\left(\sum_{j=1}^{n} \log(w_j+1)\right)$$
$$\leq O\left(w_1 \log\left(1+\frac{1}{w_1}\right)\right) + O\left(w_2 \log\left(1+\frac{1}{w_2}\right)\right) + \ldots + O\left(w_{n-1} \log\left(1+\frac{1}{w_{n-1}}\right)\right)$$
$$+O\left(w_n \log(w_n+1)\right) + O\left(\sum_{j=1}^{n} \log(w_j+1)\right).$$

Since $\log(1+\epsilon) \leq O(\epsilon)$ and $w_n = 0$,

$$T \leq O\left(w_1 \times \frac{1}{w_1}\right) + O\left(w_2 \times \frac{1}{w_2}\right) + \ldots + O\left(w_{n-1} \times \frac{1}{w_{n-1}}\right) + O\left(\sum_{j=1}^{n} \log(w_j+1)\right)$$
$$\leq O(n) + O\left(\sum_{j=1}^{n} \log(w_j+1)\right).$$

Since each x_i remains in H from the step during which bucket b_{l_i} is processed until the step during which bucket b_{r_i} is processed, we know that $\sum_{j=1}^{n} w_j \leq Q$. We then want to find the maximum value of

$$\sum_{j=1}^{n} \log(w_j+1) = \log \prod_{j=1}^{n} (w_j+1)$$

subject to

$$\sum_{j=1}^{n} w_j \leq Q.$$

The maximum is achieved when $w_1 = w_2 = \ldots = w_n = \frac{Q}{n}$. Certainly any other solution, in particular a solution that obeys the constraint $w_n = 0$, will yield a smaller value. Thus,

$$Time \leq O(n) + O\left(\log\left(\tfrac{Q}{n}+1\right)^n\right) \leq O(n \log \tfrac{Q}{n}).$$

■

6.4 Worst-Case Optimality of the Algorithm

Lemma 5 *For every given Q, there exists a sorting problem that will take $\Omega(n \log \frac{Q}{n})$ time.*

Proof We prove by contradiction. Given some Q, suppose that there exists some algorithm, A, that sorts the numbers in $o(n \log \frac{Q}{n})$ time. Consider the following problem, which we call the set-sorting problem: we have $\frac{n^2}{Q}$ independent sets of $\frac{Q}{n}$ elements each and we want to sort all the sets. We show that if A exists, then we can solve the set-sorting problem in

$$o\left(\frac{n^2}{Q} \times \frac{Q}{n} \log \frac{Q}{n}\right)$$

time, which would contradict the lower bound on sorting: to sort $\frac{n^2}{Q}$ independent sets of $\frac{Q}{n}$ elements each, requires

$$\Omega\left(\frac{n^2}{Q} \times \frac{Q}{n} \log \frac{Q}{n}\right)$$

time.

Our algorithm for the set-sorting problem works as follows. First, for each set S_j, we find max_j, the maximum element in S_j. Then, we change the sets so that each element of set S_j is strictly greater than each element in set $S_{j'}$ for all $j' < j$. We accomplish this change by incrementing each element of S_j by $\sum_{j'=1}^{j-1} max_{j'}$. Observe that this change preserves the relative order of elements within each set S_j. Now we assign the lower bounds on ranks: each element in S_j is assigned the lower bound of $\frac{(j-1)Q}{n} + 1$. Thus, we have formulated the set-sorting problem as an input to A: we have $\frac{n^2}{Q} \times \frac{Q}{n} = n$ elements to be sorted given the lower bound on the rank of each element. Note that the output of A gives us a solution to the set-sorting problem. More specifically, to get the rank of any element in S_j we take the rank of this element as given by A and we subtract $\frac{(j-1)Q}{n}$. The summary of the algorithm follows. We denote the i-th element in S_j by $x[j, i]$.

```
for j ← 1 to n²/Q do
    max_j ← −∞
    for i ← 1 to Q/n do
        if x[j,i] > max_j then max_j ← x[j,i]
incr ← 0
for j ← 2 to n²/Q do
    incr ← incr + max_{j−1}
    for i ← 1 to Q/n do
        x[j,i] ← x[j,i] + incr
        l[j,i] ← (j−1)Q/n + 1
run A on x[j,i]'s with l[j,i]'s
for j ← 1 to n²/Q do
    for i ← 1 to Q/n do
        r[j,i] ← r_A[j,i] − (j−1)Q/n
```

Now we want to analyze the running time of this algorithm. All the loops take

$$O(\frac{n^2}{Q} \times \frac{Q}{n}) = O(n)$$

time. Recall the definition of Q. So, for algorithm A,

$$Q_A = \sum_{j,i} (r_A[j,i] - l[j,i] + 1).$$

We claim that

$$Q_A = \frac{\frac{Q}{n} \times \left(\frac{Q}{n} + 1\right)}{2} \times \frac{n^2}{Q} = O(Q).$$

Thus, the running time of the set-sorting algorithm is

$$O(n) + o\left(n \log \frac{Q_A}{n}\right) = O(n) + o\left(n \log \frac{Q}{n}\right),$$

which is the contradiction we wanted to achieve.

■

Thus, our algorithm is optimal in the sense that for any Q, there exists a sorting problem that requires $\Omega(n \log \frac{Q}{n})$ time.

7 Conclusion and Open Problems

We have shown the following results in this thesis:

1. An optimal algorithm to find all farthest vertices for every vertex of a convex polygon.
2. An $O(kn \log k)$ time algorithm to find k farthest vertices for every vertex of a convex n-gon.
3. An $O(n^2)$ algorithm to sort the distances of all the vertices of a convex n-gon with respect to each vertex of the convex n-gon.
4. A worst-case optimal algorithm to sort a set of numbers given the lower bound on the rank of each number in the sorted order.

The following open questions remain:

1. A $\Theta(kn)$ algorithm to find k farthest vertices for every vertex of a convex n-gon.
2. A proof that $\Omega(n^2)$ is the lower bound for sorting the distances of all the vertices of a convex n-gon with respect to each vertex of the convex n-gon.
3. An information-theory optimal algorithm to sort a set of numbers given the lower bound on the rank of each number in the sorted order.

Acknowledgements

First and foremost I thank my family. Without their sacrifice, love and encouragement, none of this would have been possible.

I am deeply grateful to Alok Aggarwal and Tom Leighton for infecting me with their enthusiasm for research. I thank them for sharing their knowledge, for being patient, and for giving me an opportunity to work with them. I thank Tom for not breaking my hand when I played first base. I also thank Alok for stopping by my office every day with new research problems and for wading with me through numerous incorrect versions of the algorithms. His belief in me and his constant encouragement gave me the confidence I needed to do this research.

Working with James Park has been a most stimulating and productive learning experience. I would like to thank James for the great discussions, for the 305 facts about monotone matrices, and for the great passes on ice.

I thank *Running Time* and *Execution Time* for getting me away from the lab once in a while.

I also want to thank the TOC group and its mom for making the 3rd floor such a friendly and exciting place to work.

Last, but not least, I want to thank my friends for keeping me sane and for helping me feel that there is more to life than research.

Appendix

Let $p_0, p_1, ..., p_{n-1}$ be the vertices of the simple convex polygon in clockwise order and let f_i be the number of farthest neighbors of the i-th vertex.

Theorem 1 $\sum_{i=0}^{n-1} f_i \leq 2n$.

Proof Observe that corresponding to each point p_i is a b_i, $0 \leq b_i \leq n-1$, and an e_i, $0 \leq e_i \leq n-1$, that together define a range $p_{b_i}, p_{b_i+1}, ..., p_{b_i+e_i}$ of consecutive points on the polygon not containing p_i (where the indices are taken modulo n) such that

(1) p_{b_i} is a farthest neighbors of p_i,

(2) $p_{b_i+e_i}$ is a farthest neighbor of p_i, and

(3) all other farthest neighbors of p_i (if any) are among $p_{b_i+1}, \cdots, p_{b_i+e_i-1}$.

Note that $e_i + 1$ is an upper bound on f_i, the number of farthest neighbors of p_i.

Claim 2 *The points $p_{b_0}, p_{b_1}, ..., p_{b_{n-1}}$, though not necessarily distinct, are in sorted order relative to our clockwise ordering of the polygon's vertices.*

Proof Suppose not, i.e. $\exists i, j$ such that $i < j$ but p_{b_i} follows p_{b_j} in the clockwise ordering of vertices. By the quadrangle inequality, we must have

$$d(p_i, p_{b_i}) + d(p_j, p_{b_j}) < d(p_i, p_{b_j}) + d(p_j, p_{b_i}).$$

But this contradicts our assumption that p_{b_i} is a farthest neighbor of p_i and p_{b_j} is a farthest neighbor of p_j. ■

Claim 3 *For $0 \leq i \leq n-1$, $p_{b_i+e_i}$ either precedes $p_{b_{i+1}}$ in the clockwise ordering of vertices or is the same point. In other words, the ranges associated with p_i and p_{i+1} may overlap only at the point $p_{b_{i+1}}$.*

Proof Suppose not, i.e. $\exists i$ such that $p_{b_{i+1}}$ strictly preceedes $p_{b_i+e_i}$ in the clockwise ordering of vertices. By the quadrangle inequality, we must have

$$d(p_i, p_{b_i+e_i}) + d(p_{i+1}, p_{b_{i+1}}) < d(p_i, p_{b_{i+1}}) + d(p_{i+1}, p_{b_i+e_i}).$$

But this contradicts our assumption that $p_{b_i+e_i}$ is a farthest neighbor of p_i and $p_{b_{i+1}}$ is a farthest neighbor of p_{i+1}. ■

The write-up for the proof is largely from the solution by Joel Wein to problem 4 on homework 2 for a course on computational geometry (18.409) taught in the spring of 1988 by Alok Aggarwal.

The preceding two claims imply that

$$\sum_{i=0}^{n-1} e_i \leq n.$$

Thus,

$$\sum_{i=0}^{n-1} (e_1 + 1) \leq 2n$$

and

$$\sum_{i=0}^{n-1} f_i \leq 2n.$$

■

Note that for odd n, any regular n-gon gives us an example of a convex polygon for which $\sum_{i=0}^{n-1} f_i = 2n$ since every vertex of such an n-gon has exactly two farthest neighbors.

References

[AKMSW87] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, R. Wilber, "Geometric Applications of a Matrix-Searching Algorithm," Algorithmica, Vol. 2, 1987, pp. 195-208.

[AP88] A. Aggarwal, J. Park, "Notes on Searching in Multidimensional Monotone Arrays," Proc. 29th IEEE FOCS, 1988, pp. 497-512.

[BT80] M.R. Brown, R.E. Tarjan, "Design and Analysis of a Data Structure for Representing Sorted Lists," SIAM J. of Computing, Vol. 9, (1980), pp. 594-614.

[CLR89] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, to be published in 1989.

[F76] M. Fredman, "How Good Is the Information Theory Bound in Sorting?" Theoretical Computer Science, Vol. 1, 1976, pp. 355-361.

[GMPR77] L.J. Guibas, E.M. McCreight, M.F. Plass, J.R. Roberts, "A New Representation for Linear Lists," Proc. 9th ACM STOC, 1977, pp. 49-60.

[GY83] R.L. Graham, F.F. Yao, "Finding the Convex Hull of a Simple Polygon," J. Algorithms, Vol. 4, 1983, pp. 324-331.

[KS84] J. Kahn, M. Saks, "Every Poset Has a Good Comparison," Proc. 16th ACM STOC, 1984, pp. 299-301.

[LS83] N. Linial, M.E. Saks, "Information Bounds Are Good for Search Problems on Ordered Data Structures," Proc. 24th IEEE FOCS, 1983, pp. 473-475.

[P77] F.P. Preparata, "Minimum Spanning Circle," in *Steps in Computational Geometry* (F.P. Preparata, ed.), University of Illinois Press, Urbana, 1977, pp. 3-5.

[PS85] F.P. Preparata, M.I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[T83] G.T. Toussaint, "The Symmetric All-Farthest Neighbor Problem," Comp. and Math. Applications, Vol. 9, No. 6, 1983, pp. 747-753.

[TB81] G.T. Toussaint, B.K. Bhattacharya, "On Geometric Algorithms that Use the Furthest-Neighbor Pair of Finite Planar Set," Technical Report, School of Computer Science, McGill University, 1981.